The Open University

M255 Unit 14
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Software development

Unit 14

M255 Unit 14
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Software development

Unit 14

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 0 7492 1358 2

1.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).

CONTENTS

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

In this, the final unit of M255, you will be introduced to the idea of software development – a set of activities, phases (stages) and modelling techniques which help individuals and teams create software that meets its users' requirements.

By now you will have written a fair number of programs in Java and, from what you have learnt so far, you might reasonably question whether anything other than writing code is required in order to create a finished software product. However, the development of software can be far from straightforward and, generally speaking, the larger the application, the more complex the development process will be. In order to manage such complexity, it is useful to breakdown the development of software into phases. Each phase may involve different activities and/or modelling techniques which software developers can employ as they work towards the final product.

It is important to realise just how big and complex major software projects can be. Requirements for a software system might include:

- interacting with other complex systems – for example, a banking system will need to interact with the BACS (Bankers Automated Clearing Services) system;
- supporting thousands of users simultaneously – for example, a mobile phone network;
- monitoring and controlling mechanical and electrical devices – for example, a car's engine management computer;
- responding to critical conditions that affect safety – for example, a railway monitoring system;
- running continuously for months or years at a time – for example, a system for maintaining a power station;

Even fairly small software projects may have some of these characteristics.

A large project can easily involve hundreds of people working over a period of years to produce programs millions of lines long. In these circumstances software development must be a highly systematic and professional activity, or there is little chance of writing programs that work correctly and meet the customers' requirements.

In this unit we describe some of the techniques used in software development. You will not be learning about these techniques in detail, rather the intention is to give you the flavour of some of them, to allow the work you have done in earlier units to be set in a wider context.

Section 1 describes what is meant by the term 'software development', then, in Section 2, you will look at what phases may be involved in software development. In Sections 3 and 4 you will look at the modelling techniques that are employed when designing software, while Section 5 contains a discussion of how the various phases of software development can be combined to form a software development method. The unit ends by examining 'software engineering', a term used to refer to a wide range of topics related to software development.

1 What is software development?

When you are writing code you naturally concentrate on how to make the code do what is required. But how do you know what is required? Where does this information come from? Consider the types of question you might ask before starting to write code, for example:

- What should the software do?
- What classes will the software use?
- What information must objects record and to what messages should they respond?
- How will the user interact with the software?
- How can we test that the software works as it is supposed to?
- How easy will it be to adapt the software if things change later?

Software development is the process of getting from a customer's needs to operational software which meets those needs. It involves finding answers to the questions posed above and a range of related ones. Notice the term *development* – this implies something that emerges gradually, as part of a process, and does not happen all at once. Much of software development is about planning the software, and this involves building models described by text or diagrams, or both. You have already met some kinds of diagrams; for example, sequence diagrams and object-state diagrams.

If you were writing a very small program, perhaps as part of an exercise, you might not need to do much, if any, planning. For example, a program to accept two integers and output the larger might be something that you could write by typing the code straight into the computer and fixing any problems as you went along. You would obviously have to think ahead a bit, but you would not need to start by developing any models. If you did it would probably just slow you down without contributing anything.

But this only applies to very simple cases. After all, finding the bigger of two numbers is not that useful, since this is something we could easily do without a computer. Software which does really useful things is going to be more complicated. So consider a more complex example.

You have a friend who works in the administration of a local school and wants an application to help her do her job. In particular she would like some software that will support her in fulfilling the following tasks.

- For a given form (group of pupils), list information about its pupils and its teacher.
- Record the enrolment of a new pupil into a form.
- Provide the name of the teacher with the most pupils in their form.
- Provide the name of the oldest pupil in a form.

She gives you the following information.

- A record is kept of each pupil's name and date of birth.
- All pupils are aged between 4 and 18 inclusive.
- Each teacher's name is recorded.
- Each form has a name (e.g. 'Form 1b'), contains up to 10 pupils and is taught by a single teacher.

Naturally you want to help your friend, but how will you start? Of course you might try to write the code as you went along, as for the simple program we described earlier to

compare two integers. Unfortunately this approach will not work any longer. True, you have a description of what the software has to do, however there are many questions that must be answered before any code can be written. For example.

- What classes will you use?
- What Java class libraries are needed?
- What instance variables and methods will you define for any new classes?
- How are the classes related to one another?
- When the program runs, what objects will exist, and how will they be created?
- What messages will be sent, and to what objects?

You might be able to make a stab at the answers to some of these, but we hope you can see that, even with this relatively simple example, there is almost no chance of writing some working software without doing some careful planning.

To demonstrate various aspects of software development in this unit we have developed such a school management application, and you will start to explore it in the following activity.

ACTIVITY 1

Launch BlueJ, open the project called Unit14_Project, and then open the OUWorkspace. In the Code Pane execute the following code:

```
new SchoolGUI();
```

You should be presented with the user interface shown in Figure 1.

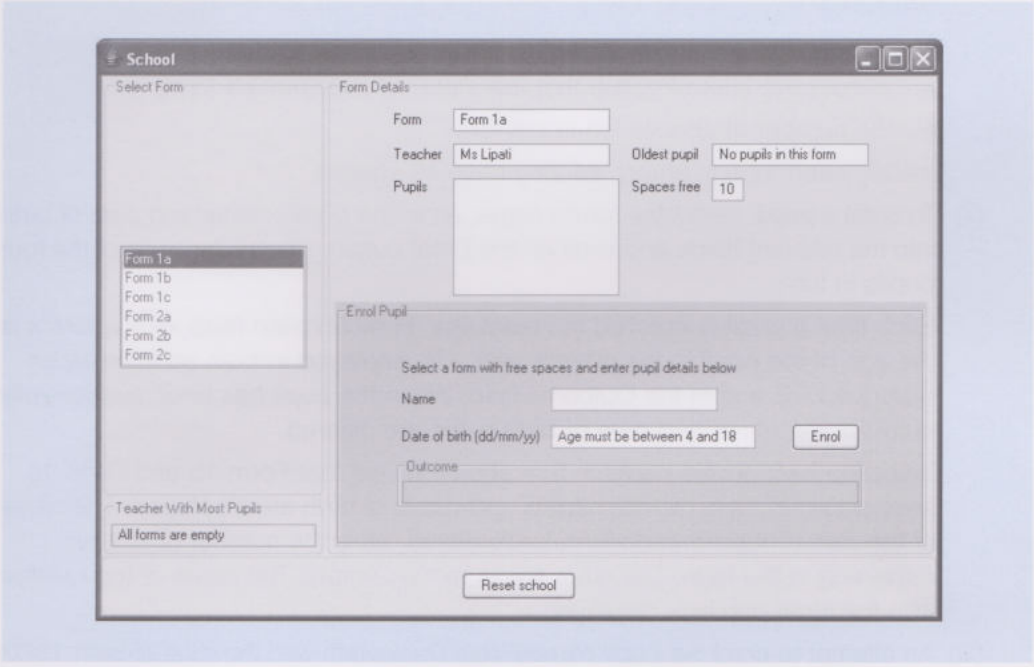


Figure 1 A simple graphical user interface for the School Management application

1 Using the user interface carry out the following tasks.

(i) View the details of each form.

(ii) Enrol, into the form named Form 1b, pupils with the following details:

Rosie Webster, who has the date of birth: 24/12/00;

Chesney Brown, who has the date of birth: 05/05/01;

David Platt, who has the date of birth: 25/12/00.

Enrol, into Form 1c, the following pupil:

Sophie Webster, who has the date of birth: 04/11/00.

What do you see in the user interface as a result of your actions? View the details of Forms 1b and 1c.

(iii) Try to enrol pupils with the following details into Form 1a:

Vera Duckworth, who has the date of birth: 12/03/45;

Joshua Peacock, who has the date of birth: 08/04/19;

What occurs in the user interface as a result of your actions?

Note that the software interprets the year 45 as 1945 and 19 as 2019.

Do not enrol any more pupils into the school yet. If you do so accidentally, or if you make a mistake and enter any incorrect details, click on the Reset school button to restore the application to its initial state. For simplicity, our basic application allows no other way of correcting errors.

2 Look briefly at the classes in the project. There is no expectation that you will understand all the details. Which classes in the project would you expect to have instances corresponding to real-world entities (i.e. 'things' in the real world)? Of the classes which do not correspond to real-world entities, what are their purposes?

DISCUSSION OF ACTIVITY 1

1 The widgets in the user interface should be familiar and hopefully the tasks were straightforward.

(i) To view the details of each form, select the form's name in the list of forms. As well as the form's name you should have found that the following are initially displayed:

- the name of the form's teacher;
- a message informing you that there are no pupils in the form;
- the number of spaces left in the form.

Initially each form is empty, so they have 10 spaces.

Recall that a form contains up to 10 pupils.

(ii) To enrol a pupil, select the form's name, enter the pupil's name and date of birth into the relevant fields and click on the Enrol button. Do this for each of the four pupils in turn.

Each time a pupil is enrolled the message 'Pupil enrolled (age x)' – where x is the age of the pupil in the current year – is displayed in both an Information dialogue box, and in the Outcome field. When the pupil has been successfully enrolled, the name and date of birth fields are cleared.

Selecting each form's name in turn should reveal that Form 1b and Form 1c have pupils in them, whose names and dates of birth are displayed. The name of the oldest pupil in each form is displayed, as is the number of spaces remaining in the form. The other forms remain empty. The name of the teacher with the most pupils is displayed.

Recall the restriction that the school only accepts pupils aged between 4 and 18.

(iii) An attempt to enrol the pupil named Vera Duckworth with the date of birth 12/03/45 results in a Warning dialogue box carrying the message 'Pupil too old! (age y)', where y is Vera's age in the current year. The error message is also displayed in the Outcome field. The name and date of birth fields are not cleared.

An attempt to enrol the pupil named Joshua Peacock with date of birth 08/04/19 results in the error message 'Pupil too young! (age -1)'.

Remember from *Unit 1*, that the domain model is that part of the software that simulates the part of the real world with which we are concerned (and is not directly concerned with how communication with the user is achieved).

2 `Form`, `Teacher` and `Pupil` objects correspond to real-world forms, teachers and pupils, respectively. You can deduce this from the names of these classes, and from the class comments. These classes constitute the domain model.

An object of the class `SchoolCoord` does not correspond directly with a real-world entity but is used to handle communication with the user interface and to coordinate the interaction between forms, teachers and pupils – much like the `BarnDanceCaller` class you encountered in *Unit 7*.

`SchoolGUI` contains the code for implementing the graphical user interface. Much of the code for this class was automatically generated using a more powerful IDE than BlueJ called NetBeans (a widely used IDE from Sun Microsystems). Since the code is automatically generated it is rather verbose, and in places does not conform to M255 coding guidelines. You do not need to understand the code.

The `M255Date` class does not form part of the domain model, it is simply a utility class that enables the School Management application to handle representations of dates easily.

SAQ 1

Why is it appropriate to refer to the School Management application as an *application* rather than a *program* or a *software system*?

ANSWER.....

As you learnt in *Unit 1*, it is not a program since it does not follow the pattern of: input data – process data – output data.

It is not, strictly speaking, appropriate to refer to it as a software system either, because the software really is not large enough to be described as such – the software is designed to run on a single computer and is not comprised of a number of subsystems or applications.

The School Management application turns your computer into a specialised computer, one to manage schools – therefore it is best described as an application.

The previous activity introduced you to a much larger example of software than you have previously seen in M255. Now imagine you were dealing with a complex *system* such as the following.

Iridium satellite system

In 2001 the Iridium satellite system, initiated by Motorola and managed by Boeing, became operational; the culmination of several years of software development. Iridium is a satellite-based communication system enabling wireless communication (for example, using mobile phones and pagers) around the world, even in remote areas. Software was developed to enable communication between mobile phones and land-based communication lines via sixty-six low-orbit satellites. This project, involving object-oriented software development processes and programming languages, produced more than 15 million lines of code.

Not only was the amount of planning for Iridium huge, but it is impossible to imagine a single programmer being able to create the system – in fact hundreds of software developers were involved. This raises another set of issues. How can a team of people succeed in working collaboratively on complex projects? How can their activities be coordinated? How do they communicate with one another?

In the next section we will start to answer these questions.

2 Development phases

This use of 'client' should not be confused with 'client' as an object in a client–server collaboration. However, the relationships are analogous: the client here is requesting a service of the developer.

In this section you will learn about the main software development phases that help software developers progress from a description of the requirements by the **client** (the person or people commissioning the software) to a deliverable working system.

Humans achieve many complicated tasks through following, consciously or unconsciously, a process of smaller, more manageable 'planning' stages. Consider the construction of a building. A process involving several levels of planning and modelling (creating different architectural plans, for example) is carried out to organise the construction engineers' thoughts (and those of their client), before any part of the building is actually constructed.

SAQ 2

Consider the task of going on holiday. How might this be successfully organised through a succession of stages, each planning some aspect of the trip?

ANSWER.....

You might begin by thinking, 'Let's take a winter break in the sun'.

Then you might visit travel agents, collect brochures, go online and consider possible dates and costs.

Next you might take decisions about where and when to go, make reservations and book leave.

Finer details are then sorted out, such as how to get to the airport, what time to get up on the day you leave, and who will feed the cat.

Finally the plan is put to the test and you set off on holiday.

The task of creating software similarly benefits from being accomplished through a systematic succession of smaller, interlinked stages, or **phases**, each consisting of different activities, and each building on the previous phase. The task of going from a description of software requirements to a collection of software objects sending messages to one another is a large and complex one, which can very easily go wrong (or may not even be possible at all) if attempted in one step. The task needs to be broken down into smaller phases that are easier both to manage and to carry out.

In software development the initial focus is usually to get an overview of the required software. That is the developer concentrates on planning the overall structure of the software and not on smaller details. As the project progresses, more detailed aspects of the software are considered. Thus, the production of what will eventually be a complex application or system is made manageable by following a development process that considers appropriate levels of detail at appropriate times. This can be thought of as moving through different levels of **abstraction** as more and more detail is added to the plans.

Painting and programming

> There is no essential difference between the way in which a painter plans and 'implements' a picture and the way in which a programmer plans and implements a program.....
>
> (In a recent exhibition).....there was one vast, unfinished canvas that revealed exactly how (the artist) had worked on it. He had sketched in the major structure, some parts completely finished, others only partly painted – exactly how a good programmer writes a program.....The processes of abstraction, visualisation and realisation are the same, just the application area is different.

Excerpt from Marshall, L.F. (1992) 'They all laughed at Christopher Columbus', in *Proceedings of the Women into Computing 1992 National Conference – Teaching Computing: Content and Methods*, Keele, UK.

A systematic development process also has the advantage that more than one person can be involved. If there is good communication between those involved, meaning not only that they talk with one another but that the scope and results of each activity are clearly set out, then allocating people to different phases enables the distinctive skills of individuals to be combined.

The object-oriented software development phases can be described as follows.

- **Requirements specification**. This involves eliciting and analysing the client's wishes in order to produce a detailed and complete specification of the requirements of the software in terms of its functionality. The requirements specification document sets out, as precisely as possible, what is required of the software. In a professional context it can form the basis for the contract between the developers and the client.
- **Developing a structural model**. Here the requirements are analysed to determine the classes and connections between them that are appropriate for the work context the software is being written for. Hence this stage defines a structure for the software. Since, in object-oriented software, objects often correspond to real-world entities, this stage starts with the creation of a model of the key features of the real-world situation within which the software is to operate.
- **Designing dynamic models**. The design of dynamic models enables the decisions to be made about what interactions among objects will achieve the tasks required of the software.
- **Developing a user interface**. This phase involves both design of the user interface and determination of how it will communicate with the domain model.
- **Detailed design and implementation**. At this stage decisions are taken as to which existing classes can be reused (from previous projects or class libraries) and what programming constructs are appropriate, as well as writing the actual code.
- **Testing**. This involves not just testing the final product but testing at each stage. Testing ensures that the software produced relates correctly to the previous stage and to the requirements.
- **Maintenance**. The aim of the maintenance phase is to keep the software working to the satisfaction of its users. It may include tasks such as:
  - fixing emerging problems;
  - fine-tuning the software to improve its performance;
  - enhancing the software by adding extra facilities.

Traditionally, a non-object-oriented approach to software development was considered to involve the following phases.

- Requirements specification. As above.
- **Analysis**. Involves analysing the specified requirements and expressing, in computing terms, *what* the software should do.
- **Design**. Involves deciding *how* the software will meet the specified requirements.
- **Implementation**. Involves translating the design into program code (of some suitable programming language).
- Testing. As above.
- Maintenance. As above.

However, when following an object-oriented approach to software development the distinction between analysis and design becomes blurred. While it is still important to distinguish between *what* the software has to do (analysis) and *how* it is to be achieved (design), the activities of analysis and design can be quite closely interleaved. In analysing the real-world tasks the software has to carry out, it is natural to think in terms of objects (because the structure of object-oriented software often resembles the real-world entities the software is concerned with). Thus, at an early stage the developer will consider not only what tasks the software is required to carry out but what objects will participate in the achievement of these tasks.

In the next section we will investigate how diagrammatic models are used during structural and dynamic modelling, to both plan the software and as a means of conveying design decisions to other members of a software development team.

3 Models

A **software model** is a plan: an illustration or description of the software, or of part of it, which emphasises certain aspects and omits others (i.e. it is an abstraction). A good analogy is a map of the London Underground, used by travellers moving between stations in the underground railway system. Such a map is shown below.

Figure 2 Map of the London Underground

The map is a representation of the London Underground system: it does not show the precise geographical layout of the lines or how the tunnels are constructed, and it does not show the location of toilets or where tickets are collected. The map is an abstraction and what it does show is a stylised description of the topological relationships between stations and connecting lines – the only information required by underground travellers to plan their route. It is a *model* of the underground system. Any information about ticket machines, toilets, and so on, would only clutter the map and make the task of finding a route through the underground system more difficult.

Similarly, the models used at different points in the software development process highlight information that is relevant at that point and suppress information that is irrelevant (i.e. the models are produced with an appropriate level of abstraction). As development progresses the level of detail in the models increases.

3.1 Modelling and diagrams

On an individual level modelling helps organise thinking about what might be a very complex task. In the context of a team working on a project, using models promotes the sharing of ideas and the successful division of tasks. For example, the design and the implementation (the actual programming) might involve different people. The designer

can hand over to the programmer a set of models representing the part of the software to be implemented. The designer need have no knowledge of the precise implementation details that the programmer may introduce; similarly the programmer need not be aware of how the designer came up with the designs. The models represent the information they need to share, and therefore constitute an important part of the communication between them.

Expressing a model using a diagram has several advantages over textual descriptions.

1 A diagram is a concise, abstract form of communication amenable to emphasising certain features and suppressing others.

2 A simple diagram can often be understood by someone inexperienced in computing (such as the client commissioning the software, or a future user of the software), whereas a textual description might not.

3 In an object-oriented approach objects begin to be identified right from the start of a project. This means diagrams involving these objects can evolve seamlessly as they incorporate increasing levels of detail through the development process. In other words, the same kinds of diagrams can be used throughout, lessening the cognitive load on the developer.

The diagrammatic modelling techniques we will look at in this unit are based on a popular modelling language called **UML (Unified Modeling Language)**.

3.2 UML

Note the American spelling of Modeling in UML.

UML (Unified Modeling Language) is an example of a **modelling language** based on diagrams. A modelling language specifies how models should be constructed so that the meaning of the model is unambiguous. It is not a *method* for developing software, but a way to produce models that could be used in different methods of software development.

Think of a language for human communication. It has:

- a vocabulary (the elements of the language);
- a grammar (the valid ways in which its vocabulary can be combined);
- semantics (what each valid combination of vocabulary means).

Similarly a modelling language has *modelling elements* (particular styles of boxes and lines, for example) and *conventions* (that prescribe what combination of elements in a diagram is valid, and that allow the meaning of a valid combination of modelling elements to be interpreted). Thus, a modelling language such as UML enables the construction of meaningful diagrammatic models of proposed software.

The rise of UML

As object-oriented programming grew throughout the 1980s and 1990s, so too did the number of modelling languages used for discussing and recording software development. From the proliferation of modelling languages one could be selected, or adapted, to suit a particular project and the people working on it. Those intimately involved in a project understood the kinds of models used, but there was no guarantee that anyone else would. Someone wanting to reuse part of the design at a later stage (for example, to implement the software in a new programming language) may have had the overhead of first getting to grips with an unfamiliar modelling notation. Reusing and even simply discussing designs was made difficult by not having a consistent and shared means of describing them.

In the late 1990s there was an attempt to establish a standard modelling language and rules for using it. Eminent software developers worked together to unify the confusing variety of existing modelling languages, resulting in proposals to a standard-setting body called the **OMG** (**Object Management Group**) for a single modelling language called UML. A **UML standard** was then set by the OMG that specified diagram elements and notation, how they could be combined, and what they meant.

The OMG is a consortium of computing companies that exists to facilitate communication within the computing industry and promote product interoperability (particularly in the area of object-oriented related software).

The UML standard is evolutionary, in the sense that there has actually been a series of standards, each building on the previous as software developers place new demands on models. At the time of writing the current UML specification is UML 2.0.

UML is a vast and, in places, highly complex language – in this unit you will meet a very small subset of the diagrams available. This is actually typical of a software project; although most professional developers have a general understanding of the expressiveness of UML, most projects will require them to work with only a limited range of diagrams.

Though UML is generally acknowledged to have made significant contributions to software development, it is also accepted that its necessary rigour makes strict adherence rather cumbersome. In particular, when using diagrams to explore different design possibilities, UML is often not strictly adhered to. Developers using UML for informal peer discussions will not see the benefits of, for example, remembering to use the right kind of arrows all the time, and they may annotate, or otherwise alter, a UML diagram to suit their own needs.

So long as the main features of diagrams follow UML, small variations tend to be unproblematic. This use of **UML-type diagrams** (i.e. ones that vary slightly from the standard) rather than **strict UML diagrams** is generally considered acceptable.

Exercise 1

A group of friends who have some experience of object-oriented software development are working together to create an application for managing their local football league. The application will undertake various tasks, including providing information about each team (for example, who the manager is) and about matches that the teams play amongst themselves in a season (who plays who, who has won the most matches etc.).

(a) State the main advantages of the friends following a planned development process.

(b) Give two reasons why it will be a good idea for them to use UML.

Solution

(a) The advantages of following a planned development process are, first, that the complexity of the application would be easier to handle, and the development made simpler.

Secondly, the planned process would allow workload to be shared, and skills put to best use, by allocation of different people to different tasks.

(b) Any two of the following are valid reasons for using UML.

- The use of models based on UML means that the group would have a consistent and unambiguous means of communication.
- Using UML would enable analysis of different possible plans for the application.
- As the group would be following an object-oriented development process then essentially the same kind of diagrams could be used throughout, reducing the number of different types of diagram involved and simplifying the process.

- UML diagrams are useful for producing diagrams at an appropriate level of abstraction (allowing detail that is irrelevant at a particular point to be suppressed).
- UML diagrams have a better chance of being understood by people other than the diagrams' creator. Some diagrams can be understood by non-computing specialists (team managers, for example, might need to know about some of the plans for the system).
- The application will be more readily understandable and reusable if UML diagrams describing the application are available.

4 Using UML

In this section we shall describe how UML is employed when developing models. We will not cover every modelling technique, nor will we go into any great detail, but this section should give you a taste of how UML is used in practice.

4.1 Modelling with class diagrams

A **class diagram** shows the structure of the proposed software, illustrating the classes that will be needed and the relationships between those classes.

You have already seen UML-type class diagrams; in fact you have seen them throughout the course. Every time you open a BlueJ project the main window displays a UML-type class diagram which shows the relationships between the classes. Up until now these diagrams have shown only one form of relationship – that of inheritance. For example opening Unit7_Project_10_sol gives the following.

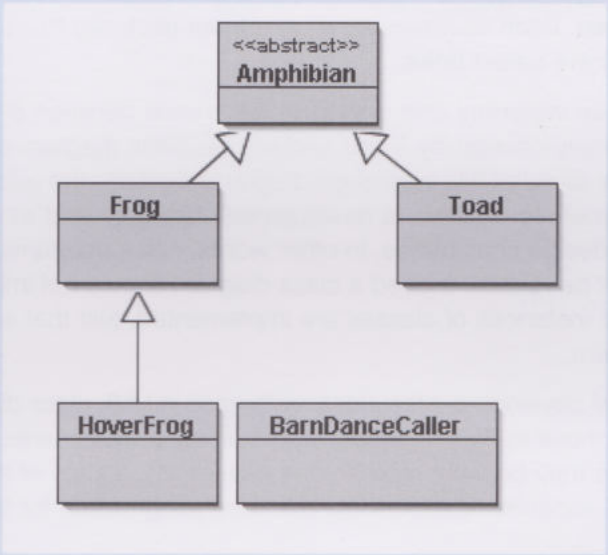


Figure 3 A UML-type class diagram displayed by BlueJ

BlueJ can display another kind of relationship in a class diagram: a uses relationship. This relationship is shown if you choose Show Uses from BlueJ's View menu. Selecting this option for Unit7_Project_10_sol gives the following.

You may need to move the classes around the BlueJ display to ensure the uses relationship is clearly displayed (and not obscured by other classes).

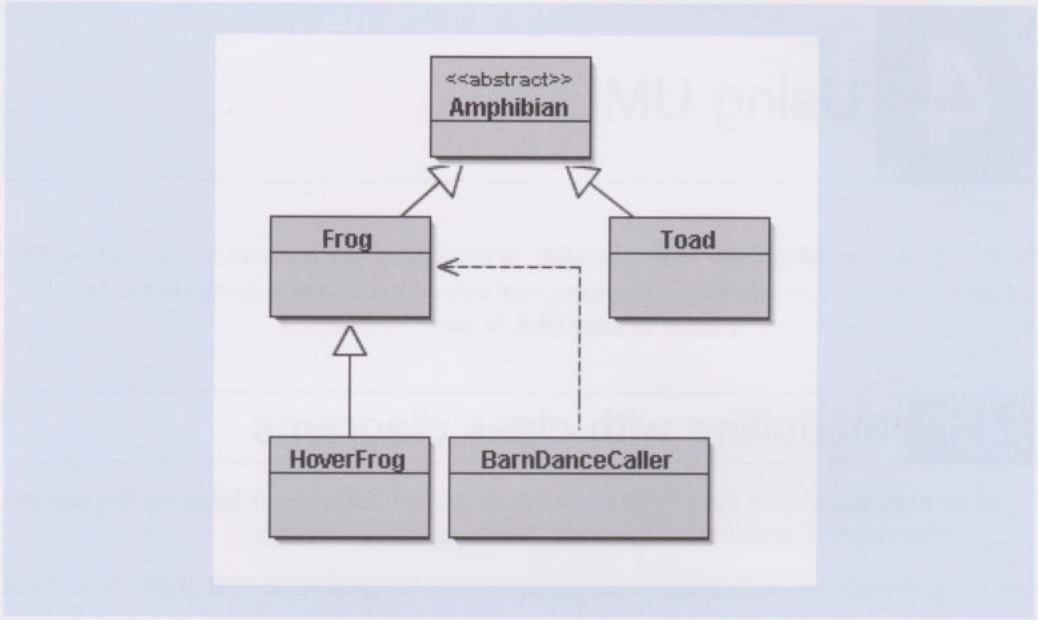


Figure 4 A UML-type class diagram displayed by BlueJ that shows a uses relationship

The uses relationship is shown in BlueJ by the dashed arrow, and (in the diagram above) indicates that instances of the `BarnDanceCaller` class have one or more instance variables that have been declared as type `Frog`. Note that while an inheritance relationship is between classes, a uses relationship represents connections between *instances* of classes. Each `BarnDanceCaller` object uses two `Frog` objects. In UML these connections are called **links**.

There is however an important and significant difference between a UML-type class diagram automatically created by BlueJ and a UML class diagram created during the design of software. BlueJ's UML-type class diagrams illustrate the relationships between existing classes whereas in software development they are used as modelling tools to explore and plan design possibilities. In other words, class diagrams suggest ideas for how classes *might* be related. Indeed a class diagram should not imply anything about *how* links between instances of classes are implemented, just that some connection exists between them.

At an early stage of development the class rectangles in UML class diagrams tend to be quite sparse (like those in BlueJ). They simply display a class name, and then as the design progresses they become more informative as the names of attributes and methods become apparent. Consider the following progression for the `Account` class.

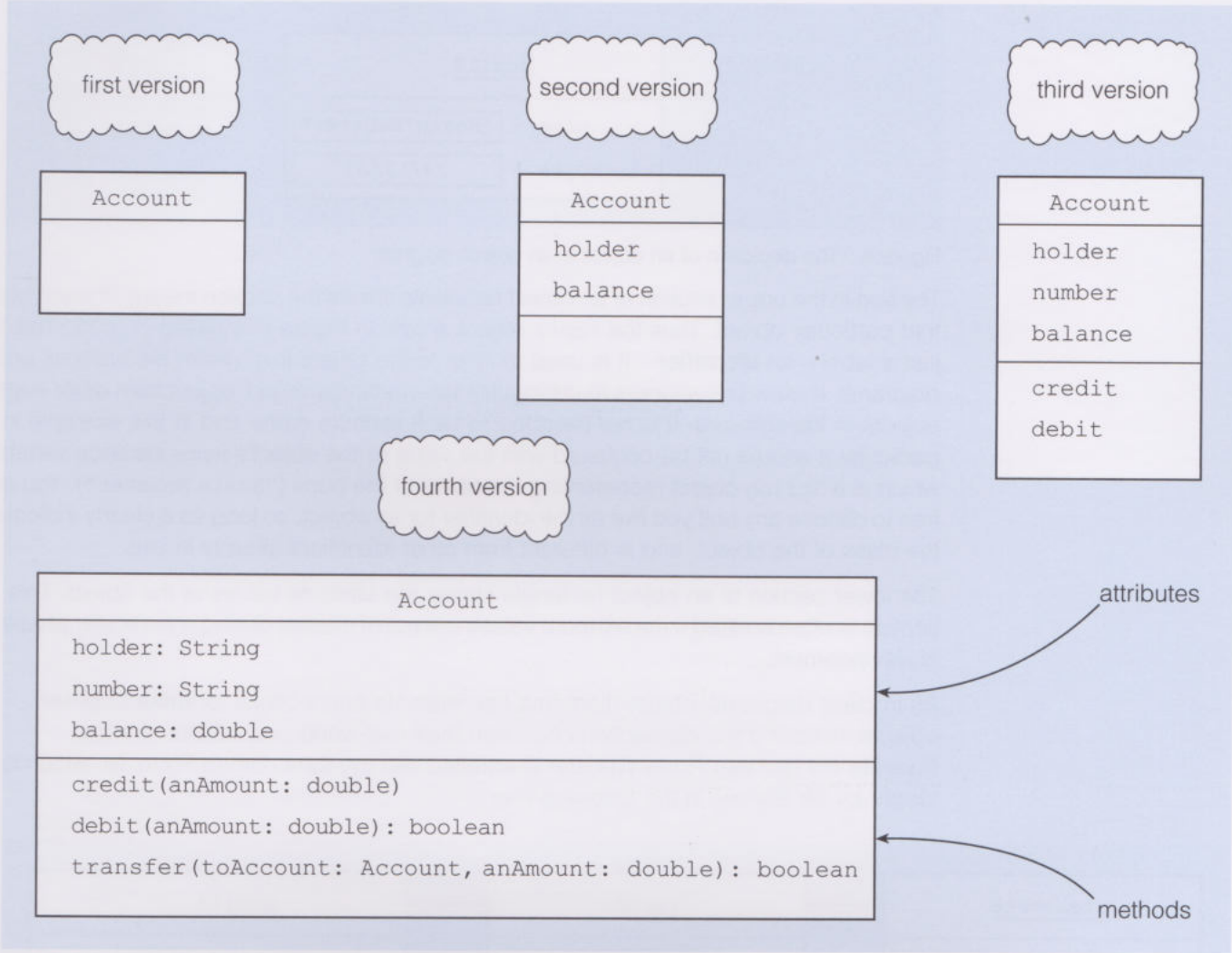


Figure 5 An evolving `Account` class

Note that in the above figure the details of attributes and methods are shown using UML syntax, not Java syntax, as UML models are not specific to a particular programming language.

Note that although in this discussion of modelling we will put all code-related labels and names into code style for clarity, a label in a UML diagram – a model –may never make it into the actual code (the implementation).

4.2 Modelling with object diagrams

Object diagrams provide another way of modelling the software under development. A UML object diagram shows the state of part of the software under development at an imagined particular point in time when it is running – a 'snapshot' if you like. In an object diagram, objects are represented by rectangles (similar to the object-state diagrams we have used throughout the course). Consider an example from the School Management application shown in the following figure.

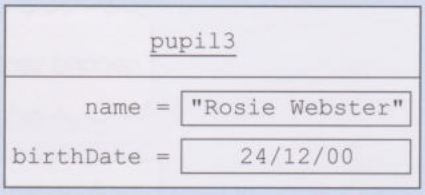


Figure 6 The depiction of an object in an object diagram

The text in the upper section of an object rectangle shows the chosen means of identifying that particular object. Thus the `Pupil` object shown in Figure 6 is called `pupil3`. This is just a label – an **identifier** – it is used to refer to the object involved in discussions and diagrams. It allows developers to distinguish this particular `Pupil` object from other `Pupil` objects in the software. It is *not* intended to be a variable name and in this example in particular it should not be confused with the value of the object's `name` instance variable which is a `String` object representing the name of the pupil (`"Rosie Webster"`). You are free to choose any text you like as the identifier for an object, so long as it clearly indicates the class of the object, and is different from other identifiers already in use.

The lower section of an object rectangle shows the attribute values of the object. This section is often omitted if the attribute values are not of interest during a particular phase of development.

As in class diagrams, object diagrams can illustrate connections, or **links** between objects, mirroring the connections between their real-world equivalents. We can illustrate the fact that Rosie Webster is enrolled into the form named Form 1b, which is taught by Mr Barlow, in the following way:

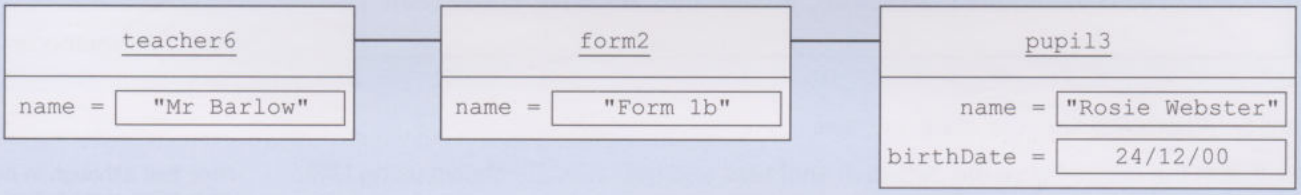


Figure 7 Object diagram illustrating links

The lines running between the object rectangles in Figure 7 illustrate links between the objects. Thus the line between the `form2` rectangle and the `pupil3` rectangle illustrates a link between `form2` and `pupil3`, and represents the fact that the form (Form 1b) corresponding to the object `form2` has in it the pupil (Rosie Webster) corresponding to the object `pupil3`.

SAQ 3

In Figure 7, what does the line between the `teacher6` and `form2` illustrate?

ANSWER.....

It illustrates a link between `teacher6` and `form2`, representing the fact that the teacher corresponding to `teacher6` (that is, Mr Barlow) teaches the form corresponding to `form2` (that is, Form 1b).

The object diagram in Figure 7 shows only part of the School Management application at run-time – it is a partial snapshot at a particular point in time. The full running application would contain many more objects and links between them with the precise situation depending on the pupils, teachers and forms in the school at that time. In an object

diagram you need include only those objects that you are interested in. For example, although the diagram shows only one `Pupil` object, there may well be other pupils in the form we have called `form2`. We refer to the full complement of objects, their attribute values (that is, the objects' states) and the links between them, which constitute the running software at any one time, as the **state** of the software at that time.

Exercise 2

Extend the object diagram in Figure 7 to show that the pupils Chesney Brown and David Platt (whom you enrolled into the school in Activity 1) are also in the form represented by `form2`, whose teacher is represented by `teacher6`.

Solution

Figure 8 shows the extended object diagram. You may have used different identifiers for the `Pupil` objects.

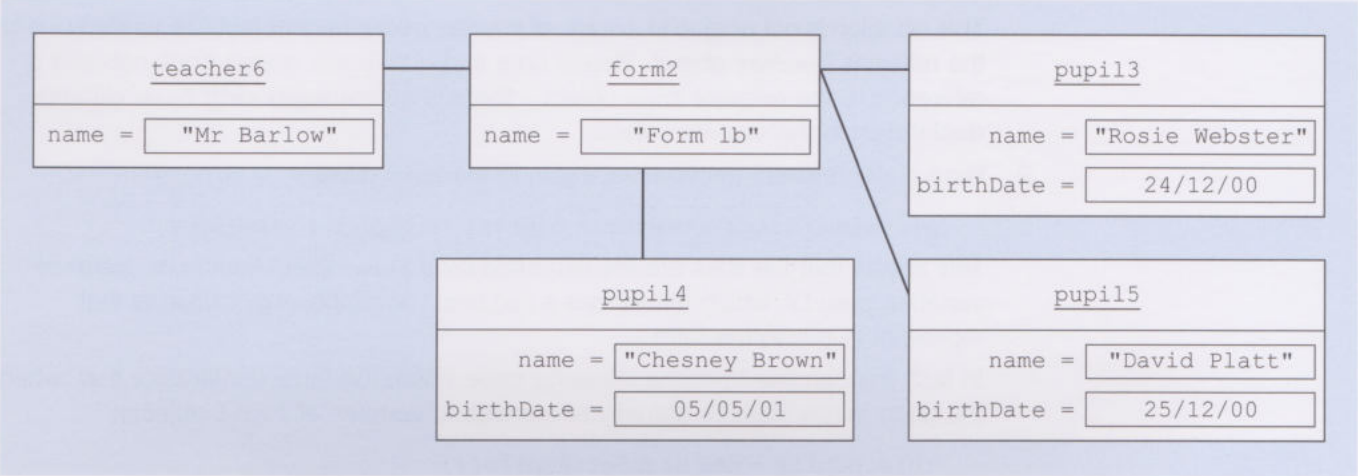


Figure 8 Object diagram illustrating `teacher6`, `form2` and its related `Pupil` objects

An object diagram created during the development of an application or system does not imply anything about *how* such links between objects are implemented, just that some connection exists. At the detailed design and implementation stage of development, different possibilities for implementation will be analysed. In the following activity you will see how this was done in the School Management application.

ACTIVITY 2

Launch BlueJ, and open Unit14_Project, which contains the classes for the School Management application. Open the OUWorkspace and run the School Management application by executing

```
new SchoolGUI();
```

in the Code Pane.

1 The running application contains, amongst other objects, the `Teacher` and `Form` objects described in the discussion of Activity 1, part 2, and SAQ 3. That is, there is a `Teacher` object with its `name` attribute set to `"Mr Barlow"`, and a `Form` object with its `name` attribute set to `"Form 1b"`. Of course, there is nothing in the application that mentions the identifiers we used (`teacher6` and `form2`) in Figures 7 and 8; remember that these are just labels used in an object diagram (which is external to the software).

Look at the source code for the classes `Teacher` and `Form`, in particular the instance variable declarations. How is the link shown in Figure 8 between `teacher6` and `form2` implemented?

2 If you followed the instructions in Activity 1 and enrolled three pupils into Form 1b, then in the running application the `Form` object we are referring to as `form2` is linked to three different `Pupil` objects. How are these links implemented?

DISCUSSION OF ACTIVITY 2

1 Although we cannot 'see' the objects in the running application, they are generated from the source code from which we can glean information about them. The following variable declaration in the `Form` class is the key here.

```
private Teacher teacher; // teacher of the form
```

This shows that the link is implemented by `form2` having an instance variable, `teacher`, which references `teacher6`.

This situation is not unique to `form2`, of course; *every* `Form` object has a reference to the relevant `Teacher` object. Please note that a `Teacher` object does not hold a reference to the relevant `Form` object – there is no corresponding `Form` variable declaration in the `Teacher` class.

2 Here is the relevant declaration, again in the `Form` class.

```
private Collection<Pupil> pupils; // pupils in the form
```

This shows that the links are implemented by a `Form` object having an instance variable, `pupils`, which references a `Collection` of the `Pupil` objects that represent pupils in the form.

In fact you can see from the following code within the `Form` constructor that, when the code is run, `pupils` actually references a `HashSet` of `Pupil` objects:

```
this.pupils = new HashSet<Pupil>();
```

Note that a `Pupil` object has no reference to the linked `Form` object.

Links between objects may be implemented by instance variables in both classes, or in just one class as is the case in our examples above. The choice of which implementation is appropriate depends on the use that the code makes of the links.

Note that, although both attributes and links can be implemented using instance variables, they are represented very differently in an object diagram. This representation highlights the fact that an object's attribute values are simple pieces of information (represented by strings, for example) that are not specific to the software under consideration, whilst in contrast its links are with other domain-model objects.

One important aspect of class and object diagrams is that, although they are expressed in software terms involving classes, methods, attributes, objects, links, etc., the client and potential users of the software usually find them easy to interpret. Therefore such diagrams can serve as a check that the developer and client have the same understanding of what classes are needed for the software, and the relationship between those classes, and instances of those classes. This is one of the main advantages of an object-oriented approach – the domain model that we are developing has a much more straightforward relationship with the real-world **problem domain** than would be the case in a more traditional approach.

4.3 Modelling with sequence diagrams

In UML, **sequence diagrams** are employed to model the software in action (i.e. at run-time), showing the message-sends involved in specific collaborations. Figure 9 shows a simple model created during development of the School Management application. It relates to the requirement for the application to provide the name of the teacher with the most pupils in their form.

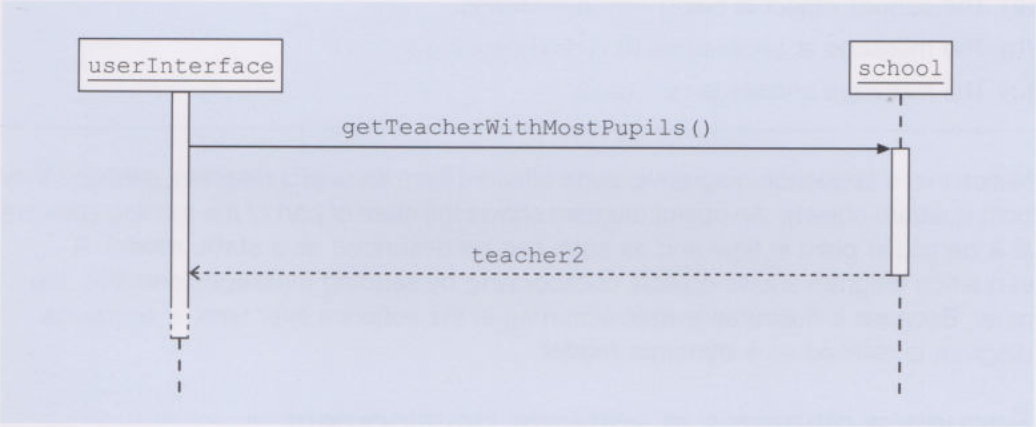


Figure 9 Getting the teacher with most pupils

You will notice that Figure 9 is a sequence diagram, of a similar style to those introduced earlier in this course (although the notation used here follows UML conventions more closely). However, there is a significant difference between how sequence diagrams have previously been used in M255, and how a diagram such as that in Figure 9 is used during the *development* of software. Whereas you have previously used sequence diagrams to illustrate the interaction between existing objects at run-time, in software development they are used as modelling tools to explore and plan design *possibilities* for how objects *might* interact. In other words, sequence diagrams in UML suggest ideas for how the future software might work.

The important features to note about sequence diagrams are as follows (some of which you will already be aware of).

- Each object in an interaction is represented by a rectangle, just as in an object diagram. This rectangle contains an identifier for the object, but no attribute values.
- Time is viewed as running vertically downwards.
- A dashed vertical line running down from an object rectangle represents the **lifeline** of that object, that is, the time during which the object exists.
- When an object receives a message, an **activation rectangle** running vertically downwards is started on that object's lifeline. This represents the period during which the object is engaged in responding to the message it has received; that is, the time during which the method invoked by the message is being executed.
- The activation rectangle for the `userInterface` object comes straight out of the object rectangle and appears 'endless' (i.e. the bottom of the rectangle is dashed). This indicates that the user interface is continuously active, always listening for events (mouse clicks, for example) caused by the user.
- A message is represented as a solid arrow.
- A message answer is shown as a labelled dashed arrow emanating from the bottom of an activation rectangle.

SAQ 4

Consider the sequence diagram in Figure 9.

(a) Which object is shown as being sent a message?

(b) What is the message?

(c) What is the message answer?

ANSWER

(a) The `school` object is being sent a message.

(b) The message is `getTeacherWithMostPupils()`.

(c) The message answer is `teacher2`.

Notice that a sequence diagram is quite different from an object diagram, although they both illustrate objects. An object diagram shows the state of part of the running software at a particular point in time and as such can be described as a **static model**. A sequence diagram shows objects collaborating by sending messages one after the other. Because it illustrates events occurring in the software over time, a sequence diagram is classed as a **dynamic model**.

Sequence diagrams in software development

In the design stages of a software development project, **scenarios** which represent typical user interactions with the software, are devised. For each scenario, sequence diagrams are created to show which message-sends will need to be exchanged between objects in the running software for the scenario to be completed. Sequence diagrams therefore form the basis for deciding which methods are appropriate for the classes of the emerging software, and what each method should involve.

Consider the following example. For the School Management application there is a requirement to provide the name of the oldest pupil in a given form. We will consider a particular scenario involving finding the oldest pupil in a particular form.

In Activity 1 you enrolled the following pupils into the form called Form 1b:

Rosie Webster, date of birth: 24/12/00;

Chesney Brown, date of birth: 05/05/01;

David Platt, date of birth: 25/12/00.

Suppose that, in the development of this application, a scenario involving pupils as described above was devised. The developers might illustrate the objects involved in this particular scenario in the object diagram in Figure 10. This shows the `Form` object `form2` (named 'Form 1b') together with the `Pupil` objects corresponding to all the above pupils in the form, which are labelled as `pupil3`, `pupil4` and `pupil5` in the figure. The application of course includes other `Form` and `Pupil` objects, as well as objects of other classes, but they are not relevant to our current investigations.

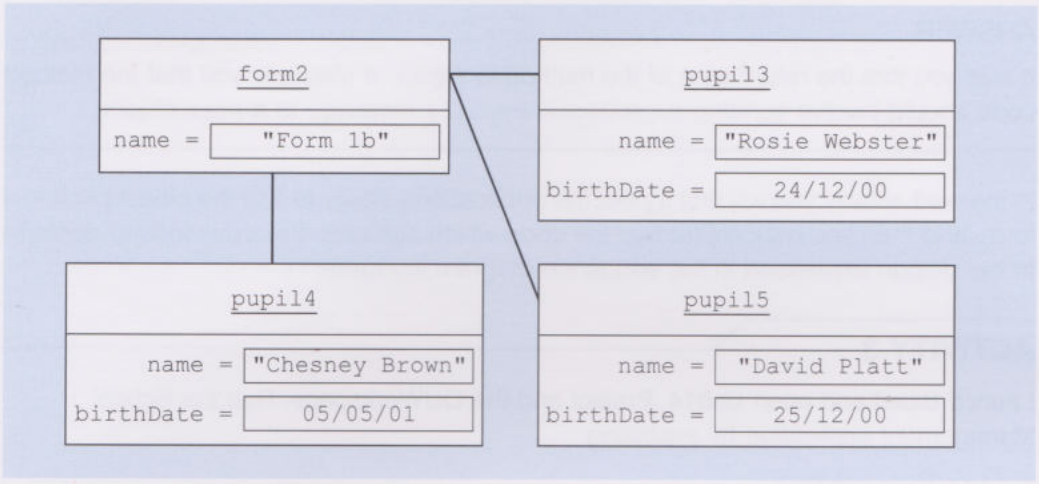


Figure 10 `form2` and its `Pupil` objects

Figure 11 shows a simple sequence diagram for this scenario, that the developers of this application might have considered, which expresses a particular design idea.

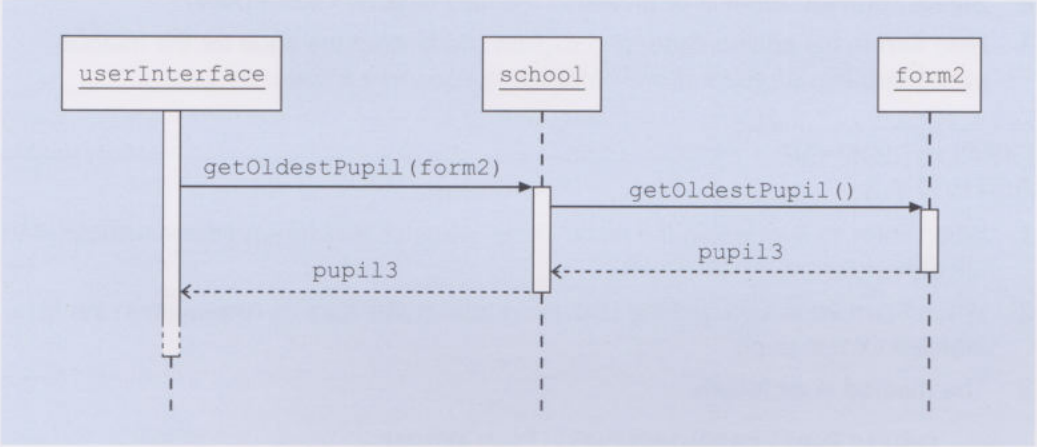


Figure 11 `school` returns a message answer

In creating the sequence diagram above the developers were expressing the idea that, as part of what the application does to get the oldest pupil in a form:

- the object `school` (an instance of the class `SchoolCoord`, which you met in Activity 1) could receive a message `getOldestPupil()` (with a particular `Form` object, here `form2`, as the argument) from the user interface, represented by the object `userInterface` (an instance of `SchoolGUI`);
- the object `school` would then send a `getOldestPupil()` message to the `Form` object, which would respond by returning the oldest `Pupil` object (`pupil3`, in this particular scenario) to `school`;
- finally the `school` object would return that `Pupil` object to the user interface.

The sequence diagram in Figure 11 is a model which emphasises a collaboration between the `userInterface` object and the objects `school` and `form2`, but neglects details such as the precise method code.

SAQ 5

Imagine that you, as a programmer, are handed the sequence diagram shown in Figure 11 by a designer. What information does the sequence diagram give you about how to code the `SchoolCoord` method with the signature `getOldestPupil(Form)`?

ANSWER

It tells you that the return type of the method is `Pupil`. It also tells you that the method code should involve sending a `getOldestPupil()` message to a `Form` object.

Do not confuse the two methods involved. The first is `getOldestPupil(aForm)` in class `SchoolCoord` and the second is `getOldestPupil()` in class `Form`. Although they happen to have the same name they are entirely distinct.

In the next activity you will first try out the application's ability to find the oldest pupil in a form, and then you will confirm that the code which achieves this does indeed conform to the design expressed in the sequence diagram in Figure 11.

ACTIVITY 3

Launch BlueJ and open Unit14_Project and the OUWorkspace. Run the School Management application by executing

```
new SchoolGUI();
```

in the OUWorkspace's Code Pane.

1 In the user interface, select Form 1b. What is displayed in the field labelled Oldest pupil?

2 Select Form 2a. What is displayed in the field labelled Oldest pupil?

3 Now turn to the source code, and explain briefly what the code for the method `getOldestPupil(Form aForm)` in class `SchoolCoord` does.

DISCUSSION OF ACTIVITY 3

1 When Form 1b is selected the name 'Rosie Webster' is displayed in the field labelled Oldest pupil.

2 When Form 2a is selected the text 'No pupils in this form' is displayed in the field labelled Oldest pupil.

3 The method is as follows

```
public Pupil getOldestPupil(Form aForm)
{
    return aForm.getOldestPupil();
}
```

The method sends the message `getOldestPupil()` to the object referenced by `aForm` (the method argument). This corresponds to the sequence diagram in Figure 11 in which the message `getOldestPupil()` was sent to the object `form2` of the particular scenario.

The sequence diagram in Figure 11 shows the collaborations between `userInterface`, `school` and `form2`. Suppose the developers then turned to considering how a `Form` object would respond to a `getOldestPupil()` method. In this particular scenario what might `form2` do to return its oldest pupil? An obvious approach would involve `form2` asking each of its `Pupil` objects in turn for their birth dates: that is, collaborating with each `Pupil` object. The following exercise asks you to consider how the developers might illustrate this idea in a sequence diagram.

Exercise 3

Figure 12 shows the first of the collaborations between `form2` and its `Pupil` objects. Complete the diagram to show the collaborations between `form2` and `pupil4` and between `form2` and `pupil5`. (You will need to refer to Figure 10 for the birth dates.)

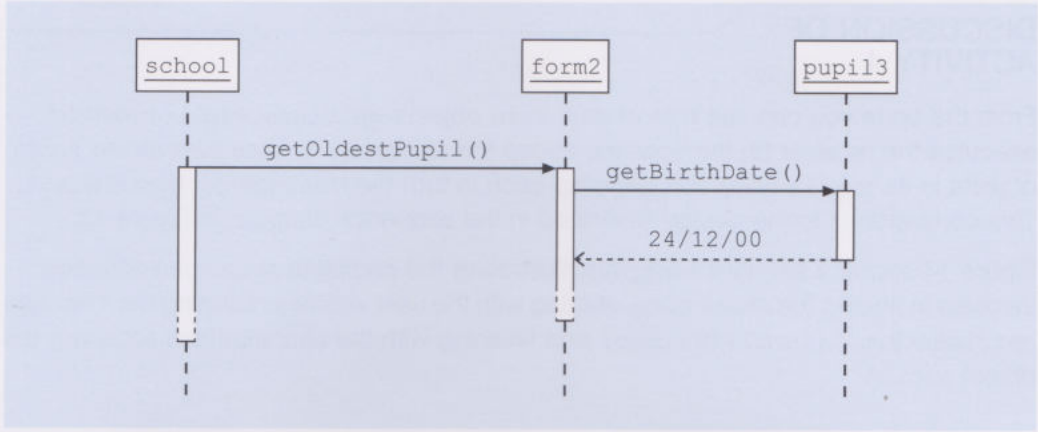


Figure 12 `form2` collaborates with `pupil3`

Solution

Figure 13 shows the expanded sequence diagram with the collaborations between `form2` and the three `Pupil` objects depicted.

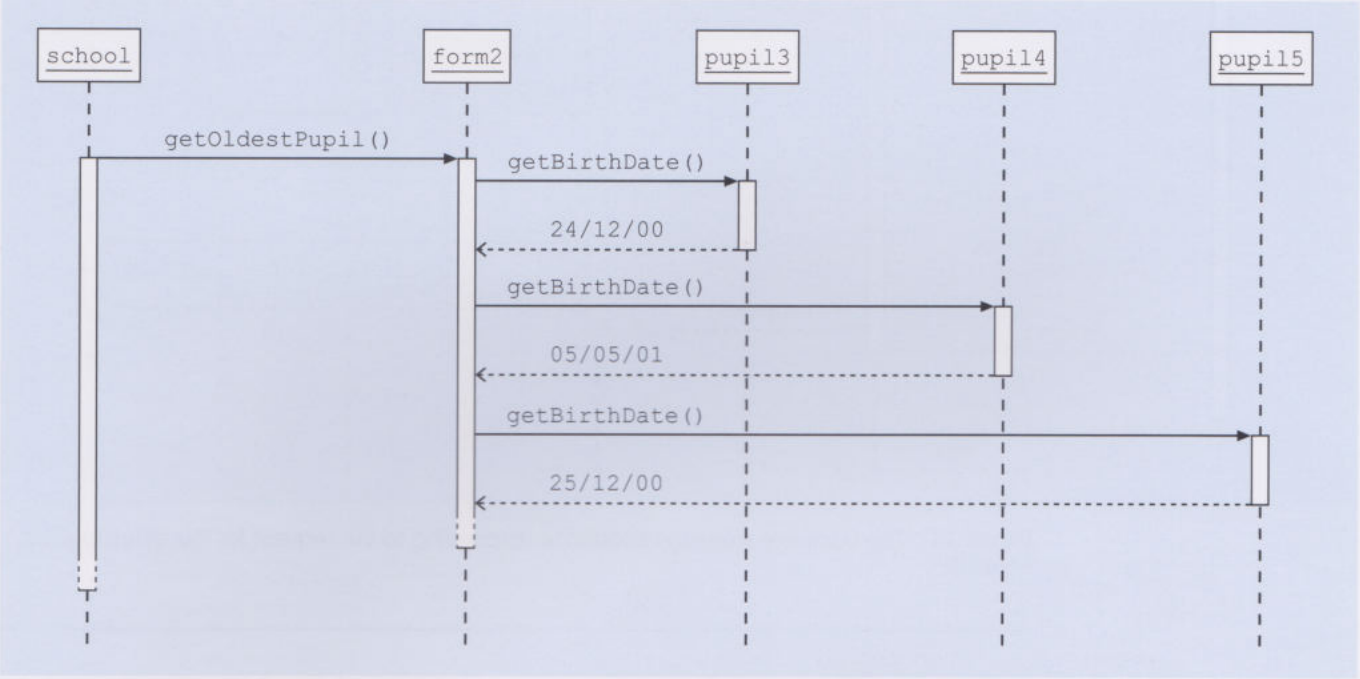


Figure 13 `form2` collaborates with each of its `Pupil` objects

In Activity 4 you will confirm that the School Management application code conforms to the design expressed in the sequence diagram in Figure 13.

ACTIVITY 4

Launch BlueJ and open Unit14_Project.

Explain briefly how the code for the method `getOldestPupil()` in class `Form` corresponds to the design illustrated in Figure 13.

DISCUSSION OF ACTIVITY 4

From the code you can see that when a `Form` object's `getOldestPupil()` method executes the receiver (in the scenario above this is `form2`) iterates over all the `Pupil` objects in its `pupils` collection, sending each in turn the message `getBirthDate()`. This corresponds to the design illustrated in the sequence diagram in Figure 13.

Figure 14 shows a sequence diagram illustrating the complete message sequence involved in finding the oldest pupil, starting with the user interface sending the message `getOldestPupil(form2)` to `school` and finishing with the user interface receiving the object `pupil3`.

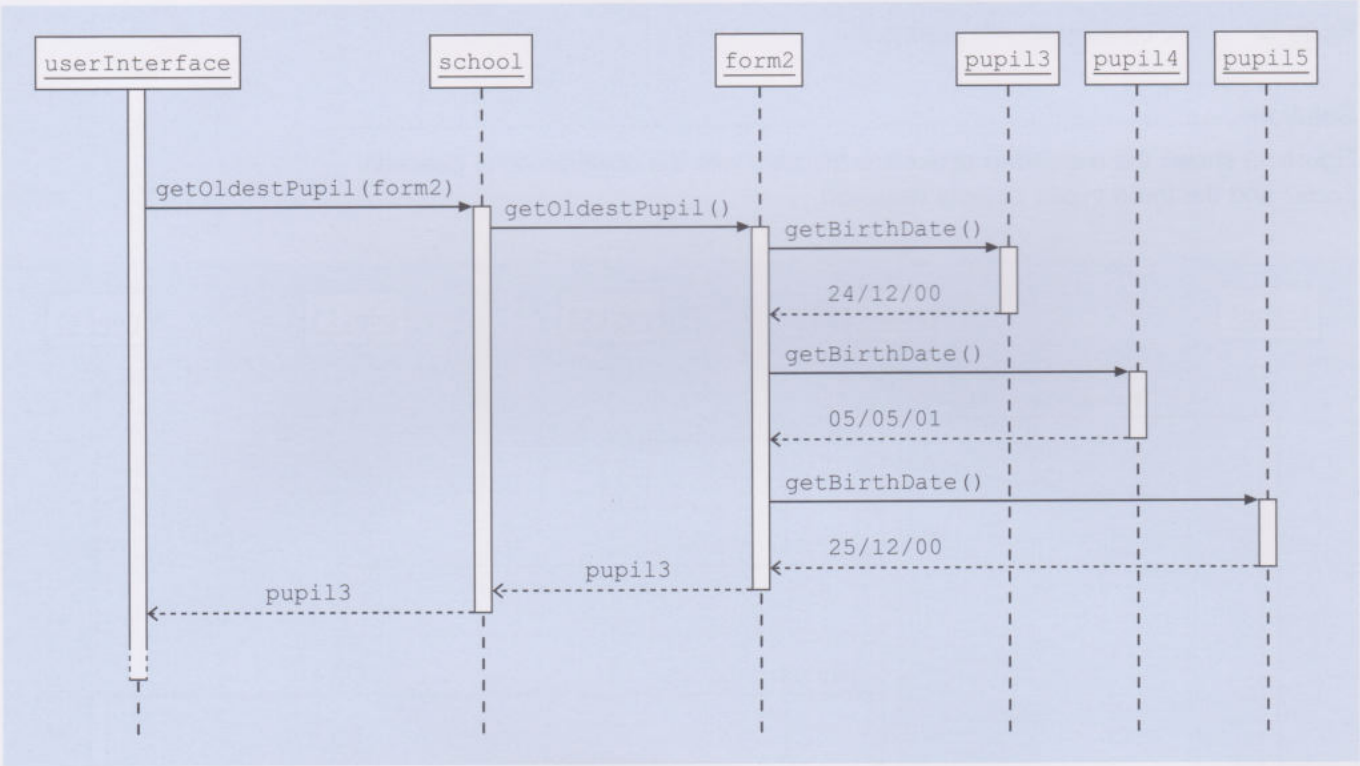


Figure 14 The complete message sequence responding to the request for the oldest pupil in Form 1b

In the next exercise you will practice what you have learnt about object and sequence diagrams, by again imagining that you are in the process of developing the School Management application. This time you will be looking at exactly the same task (locating the oldest pupil in a form) but with a different scenario.

Exercise 4

(a) Suppose that the form named Form 1c has one pupil in it – her name is Sophie Webster and her date of birth is 04/11/00. Ms Yingjie is the teacher of this form. Draw an object diagram, using the identifiers `form4`, `pupil6` and `teacher1`, to illustrate the objects that correspond to these real-world entities.

(b) Suppose that a pupil named Craig Harris enrols into Form 1c. His date of birth is 02/07/00. Extend your object diagram to illustrate the `Teacher`, `Form` and `Pupil` objects involved, choosing a suitable identifier for the additional object.

(c) Suppose that a user of the School Management application selects Form 1c in the user interface. Draw a sequence diagram to illustrate the sequence of messages and message answers that should pass through the application for this scenario, resulting in the `Pupil` object corresponding to the oldest pupil in Form 1c being returned to the user interface.

Solution

(a) The object diagram for Form 1c is as follows.

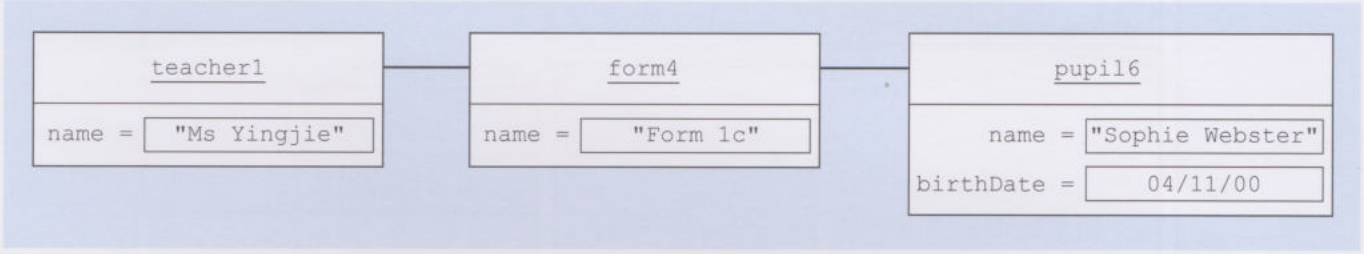


Figure 15 Object diagram illustrating Form 1c, its teacher and its pupil

(b) In our updated object diagram we have used the identifier `pupil7` for the additional object. You could have used any identifier that was different to the ones that have already been used in this unit.

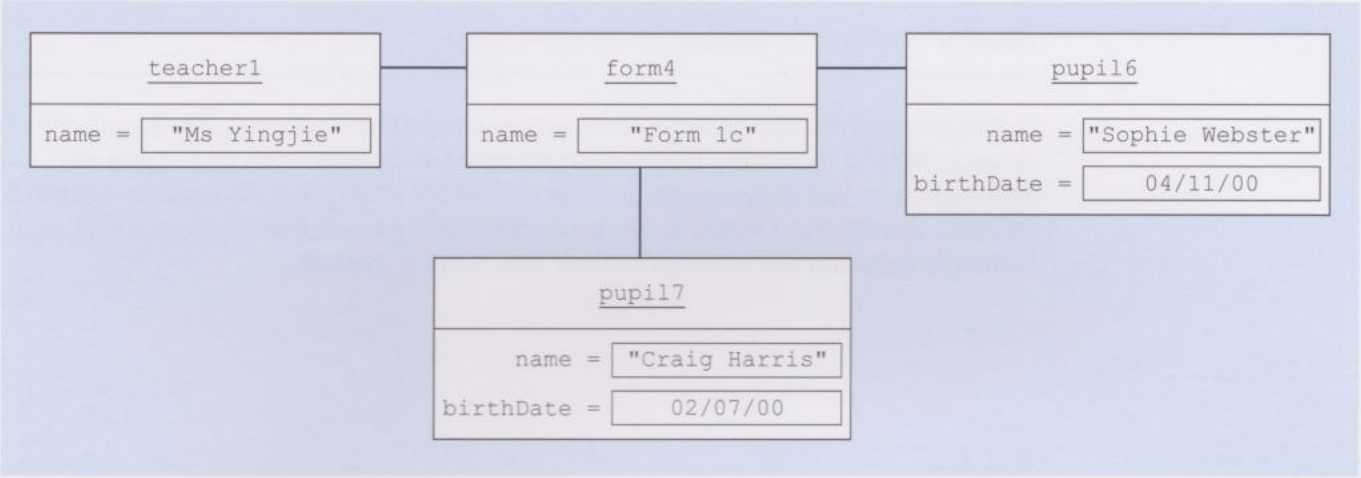


Figure 16 A new pupil in Form 1c

(c) A sequence diagram, showing the scenario in which the oldest pupil from Form 1c is obtained, is shown in Figure 17. Note that the order in which the `Pupil` objects are sent the message `getBirthDate()` does not matter.

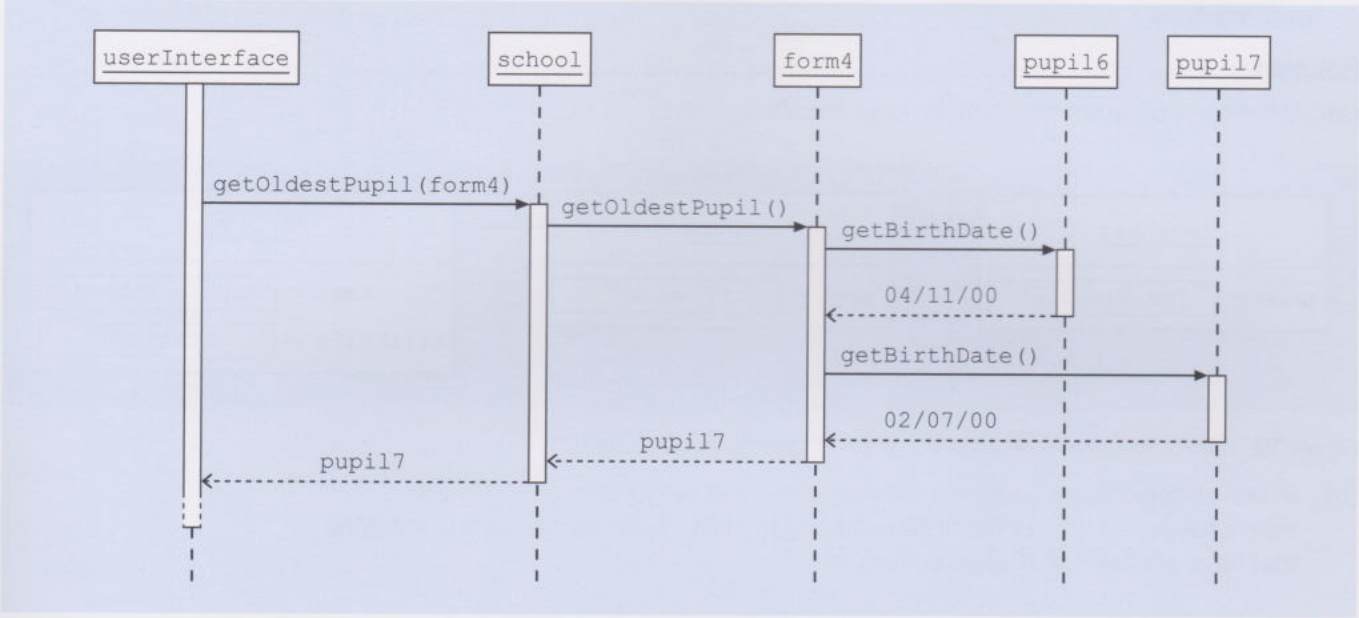


Figure 17 Getting the oldest pupil in the new scenario

In this section you have explored how sequence diagrams can be used in software development for illustrating the interactions between objects involved in particular scenarios, and how they can be used as a basis for writing method code. In Section 5 software development methods are introduced, and you will look at examples of such methods, including the waterfall method and iterative methods.

5 Software development methods

Earlier you were introduced to the concept of developing software in phases, each building upon the previous phase. This section gives an overview of how the phases of software development may be combined to form a **software development method**.

5.1 What is a software development method?

Before discussing what is meant by a software development method, it might be helpful to review briefly what has been learnt about the phases of software development.

In Section 2 we introduced the following main phases of object-oriented software development:

- requirements specification;
- developing a structural model;
- designing dynamic models;
- developing a user interface;
- detailed design and implementation;
- testing;
- maintenance.

SAQ 6

What is each of the following phases concerned with?

(a) developing a structural model.

(b) designing dynamic models

ANSWER.....

(a) Developing a structural model is concerned with determining what classes and objects (and the relationships between them) are appropriate for the requirements.

(b) Designing dynamic models is concerned with determining what interactions among objects will achieve the requirements.

It is most important to appreciate that there is no implication that the phases *must* be undertaken in a linear fashion, with each one completing before the next starts. On the contrary many different permutations are possible. A **software development method** is a particular set of phases and their activities, applied in a particular order.

At this point you may be wondering why there is a need for different software development methods. First, there is much debate, and no obvious consensus amongst practitioners and researchers, on the relative merits of different approaches to creating software. Secondly, there can be major differences between software projects, which determine which methods are appropriate. For example, a significant influence on choice of development method is the stability of the software requirements, that is, whether they can be fully determined at the outset of the project, and how liable they are to change. The requirements for an embedded system, such as a washing machine controller, or a safety-critical system controlling a power station, may be well defined

from the start, and unlikely to change. In contrast the requirements for a stock control system for a newly established business will change with the changing nature of the business. Changing requirements can require very flexible development methods.

5.2 The waterfall method

The **waterfall method** is a traditional and idealised view of software development and involves strictly following a sequence of phases. It describes development in which each phase is visited only once, and where each phase is completed before the next begins. Figure 18 illustrates this.

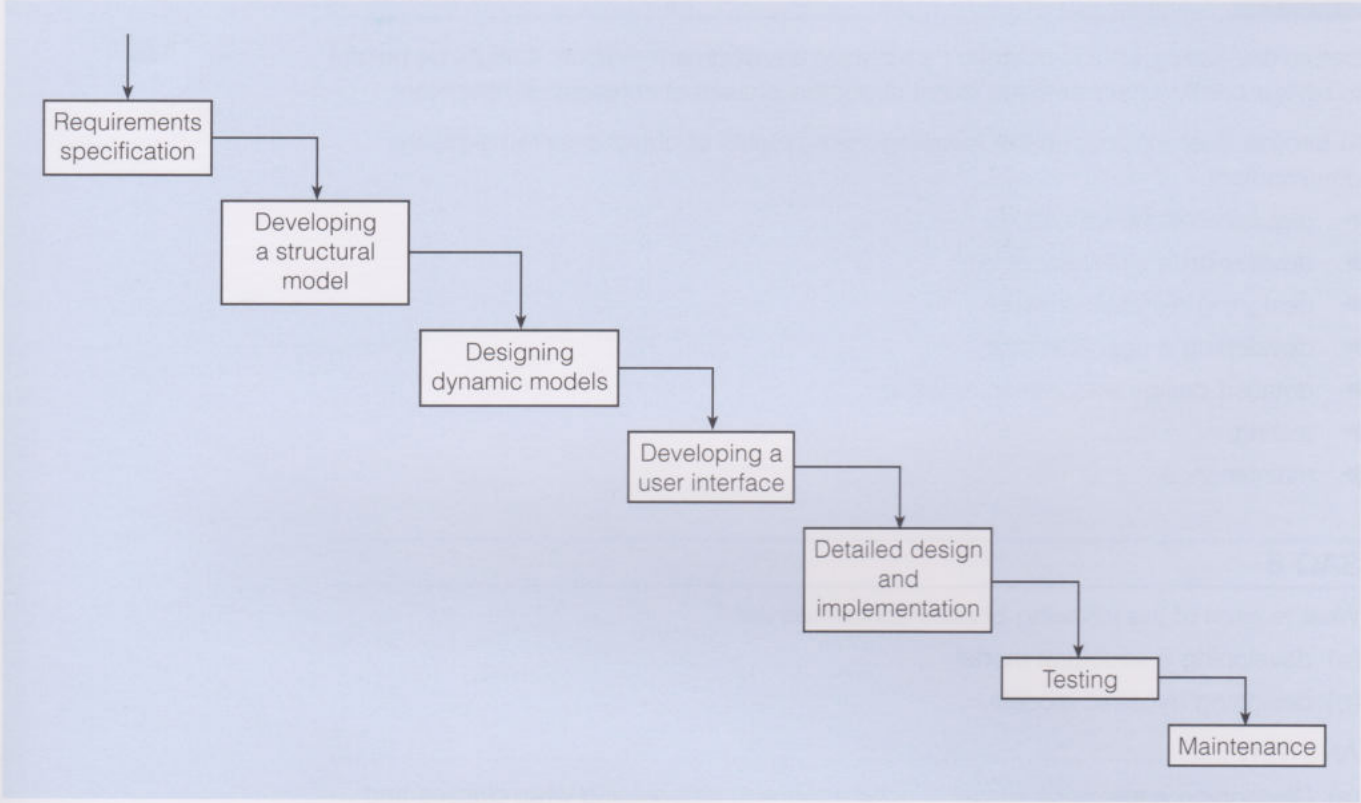


Figure 18 The waterfall model of software development

SAQ 7

Can you think why the waterfall method has been nicknamed the 'throw it over the wall' method?

ANSWER.....

The method is nicknamed the 'throw it over the wall' method since once a phase is completed it is essentially beyond the control of the developers – they may not revisit it.

The waterfall method has some advantages for the management of a project. If there are a set number of phases then we can at least try to plan in advance for the time and resources required for each phase and then for the entire project. But the method suffers from a number of problems.

1 It does not produce any executable software until the end of the project, so the client may not have a good idea of what they are getting until it is too late to make changes.

2 Testing, being at a late stage in the project may be neglected if the project overruns.

3 Errors are likely to be undiscovered until late in the project, meaning that resolving them is rushed or not done at all, or the project is delayed (with the associated problem of considerable costs being incurred).

4 The method does not countenance changes or additions to the requirements as the project progresses, but relies on all the requirements of the software being established at the beginning. This is often unachievable.

5 There is no allowance for the developers to return to a phase to revise earlier decisions.

As previously indicated many projects do not start with a fixed and unchanging set of requirements, and most developers do not make consistently perfect decisions. Thus rigid adherence to a waterfall method is generally unrealistic. Nevertheless, many projects do follow an approximation to it (deviating, for example, by allowing a return from implementation to dynamic model design when a coding problem arises), largely because its predictability aids project management. The term **predictive method** is sometimes used to describe a method largely based on the waterfall approach.

5.3 Iterative methods

Whereas a predictive method is inflexible in the face of change, an **adaptive method** of software development is able to respond to change. *Adaptive* describes ways of developing software, which not only tolerate change (to the software requirements, to ideas in the developers' minds, etc.), but which actually embrace change by building space for it into the schedule.

An **iterative method**, common in object-oriented software development, is one such adaptive method. Phases are repeated in a systematic manner, with each **iteration** (one cycle through the phases) enabling the developers to build on the work completed so far, as well as offering an opportunity for reflection and revision.

A common iterative practice is to restrict the initial development to only a small subset of the requirements. By designing and implementing just a part of what is required the developer is able to get early feedback from the client and thus reveal more quickly any problems arising from misunderstandings of, or changes to, the requirements. Once this initial version of the software has been implemented satisfactorily, additional behaviour can be incorporated by repeated iterations of the development process until eventually a version of the software that satisfies all the specified requirements is produced.

Figure 19 shows an outline of an iterative method.

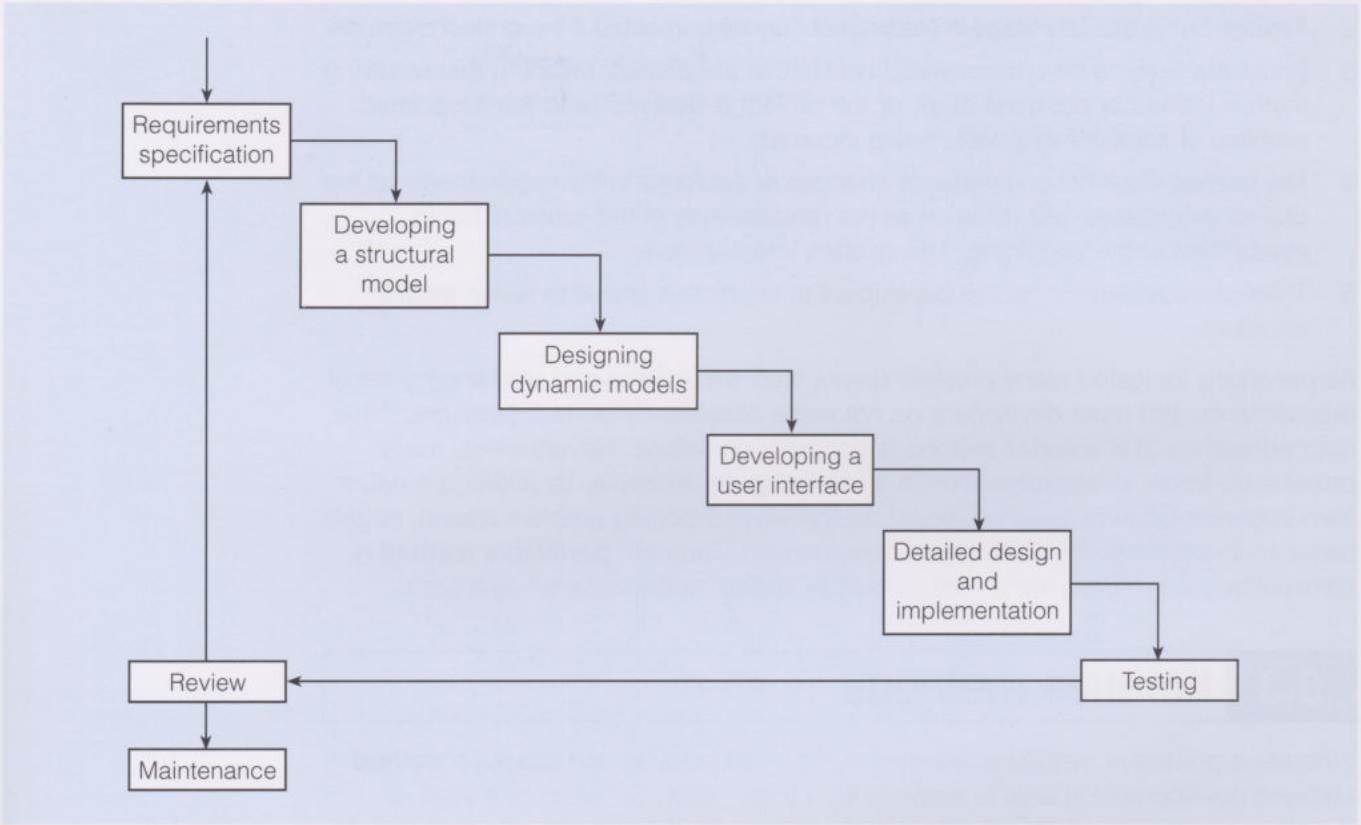


Figure 19 An iterative approach to software development

There are many variations of the iterative approach. A common one is for early iterations to concentrate on getting a satisfactory design of the *structure* of the software before going into the detailed design and implementation and testing phases.

The **review**, which is explicitly included within each iteration (see Figure 19), is a point where developers and clients can take changes into account by scheduling them into a future iteration.

SAQ 8

List some kinds of changes likely to be identified within a review.

ANSWER

Here are some of the changes you might have thought of.

- Changes in the client's requirements.
- Changes to decisions made in previous iterations – about the structure of the software, its design or its implementation.
- Changes to correct any errors from previous iterations.

In each iteration the designs and/or code are tested. Since one iteration builds on another, tests are repeated to ensure that the changes and additions made during an iteration do not damage the previous development.

Within iterative development, **prototypes** are often useful. A prototype is an early working version of the required software, or part of what is required, used to test and confirm ideas about what the software is required to do and how best to achieve this. For example, a part of the user interface (with perhaps limited or no actual functionality) may be designed and implemented so that its usability can be analysed and the results fed into the development process.

eXtreme Programming (XP)

Emerging in around 2000, the ideas of **eXtreme Programming (XP)** challenge the wisdom of developing software through carefully planned phases. Instead, XP advocates (at least on certain kinds of projects) concentrating on rapid, prototype-producing, documentation-light iterations of coding and testing. XP is an example of an **agile** development process, which prioritises the people and styles of teamworking on a project ahead of any process and documentation used.

6 Software engineering

Systematically developing software by using a defined methodology is a vital ingredient in a successful project. We noted earlier, similarities with the way in which a building is developed; in fact there are similarities with the development of engineering artefacts more generally. A succession of activities is involved, moving from a general description of the software (product) through increasingly detailed designs (engineering blueprints) to the implementation (construction). Because of these similarities to engineering physical artefacts the term **software engineering** is often used to refer to the wide range of issues connected with carrying out successful software development projects (particularly large-scale ones) using a systematic approach. Software engineering covers not only the technical aspects of building software systems, but also management issues, such as directing programming teams, scheduling and budgeting.

The theory of software engineering is a vast one, with substantial industrial practice and academic research behind it. For an idea of its extent, consider the following small sample of research areas.

- **Software development methods**. For example, which kind of method suits which kind of project?
- **Project management**. How best to manage the people, tasks, resources and finances involved in a project, so that software of an acceptable quality is delivered on budget and on time?
- **Risk analysis**. Identifying and managing the possibility of problems occurring in a project. How would a project cope if one of its developers left, for example?
- **Testing techniques**. For example, what testing strategies are best suited to what kind of project?
- **Software quality**. What are desirable qualities of software, and how can they be measured and maximised?

The rest of this section introduces you to aspects of software engineering relating to **large-scale projects**. By *large scale* we mean that the size of the project, and the complexity of detail involved, precludes the work being carried out by one or two people, and hence that the project can only be carried out by a team of people, with different individuals working on different aspects of the project.

6.1 Project failure

Developing software involves a complex process of analysing different possibilities and making choices. It is an inventive and therefore challenging activity which offers opportunities for satisfying creativity, but also for disappointment. Disappointing software is often the result of poor software development. The term **project failure** covers situations where a project is unsatisfactory in some way: it might be over budget, it might run over time, or it might result in unsatisfactory software.

Software development can, though rarely, result in software that completely or almost completely fails. You might like to pause at this point and try to recall an example of a real-life software project that has largely failed.

There are many notorious examples. You may have remembered the following.

- The Child Support Agency system put into operation in 2003. Problems with this new system resulted in a backlog of millions of pounds of unpaid support payments to single parents.
- The UK Passport Agency problems of 1999, when the introduction of a new computer system resulted in long delays in the processing of passport applications and queues of passport applicants outside the agency's offices.

Common causes of project failure

The UK National Audit Office and the Office of Government Commerce published a list of common causes of public sector project failure in 2004 (for more details see http://www.nao.org.uk/publications/nao_reports/03-04/0304877es.pdf; accessed 2 June 2006). The points (simplified in places) were as follows.

1. Lack of clear link between the project and the organisation's key strategic priorities, including agreed measures of success.
2. Lack of clear senior management and ministerial ownership and leadership.
3. Lack of effective engagement with clients.
4. Lack of skills and proven approach to project management and risk management.
5. Lack of understanding of, and contact with, the supply industry at senior levels in the organisation.
6. Evaluation of proposals driven by initial price rather than long-term value for money (especially securing delivery of business benefits).
7. Too little attention to breaking development and implementation into manageable steps.
8. Inadequate resources and skills to deliver the project.

6.2 Teamwork

When a team works on a software development project, it is usual for different people to work on different parts of the development. A professional may specialise in a particular aspect of development, concentrating on that aspect in the projects they work on. You may have heard of some of the following job titles, all of which come under the umbrella title of **software developer**.

- A **systems analyst** works closely with a client who has requested a software solution to determine if a software solution is practicable. If it is the systems analyst will then determine how such a solution would fit into the rest of the client's current business practices and how those practices may need to adapt with the introduction of the proposed software. The systems analyst will then ensure that the clients requirements are expressed in a consistent and non-contradictory form that can be understood and acted upon by the requirements analyst. To do this a systems analyst ideally needs knowledge of both the **problem domain** (the client's business needs) and software design.
- A **requirements analyst** analyses the requirements produced by the systems analyst to produce a rigorous requirements specification that can be acted upon by a software designer. They may also become involved in preliminary aspects of design.

- A **designer** takes the requirements specification and works on the design stages of a project. A designer may specialise in certain kinds of design, for example games design or user interface design.
- A **programmer** implements the code, based on the design models, testing small units of code along the way. Again, there are various specialists, such as games programmers.
- A **technical writer** is involved in developing user documentation, such as help files and user manuals.
- A **software tester** tests the software as it is being developed; for example, testing that separate units of code, perhaps written by different programmers, interact appropriately together (integration testing).
- A **project manager** plans and oversees the running of a software development project, from making an initial assessment of the risks involved in the project and allocating people to teams, to having ultimate responsibility for the decisions taken during the project and handing over the software to the client.

It is not uncommon for different software development firms to be commissioned to work on different aspects of a software project. One firm might carry out the systems analysis, another the requirements analysis, yet another the software design, and so on. With so many people involved you can see why the project manager figures large in the process!

SAQ 9

Figure 19 (in Subsection 5.3) shows the iterative software development method. Which developers might you expect to be involved at the review stage?

ANSWER.....

A review allows changes to be taken into account by scheduling them into a future iteration. Since these changes can affect the work of any one of the developers it is quite common that *all* developers are involved in a review. Certainly all those who have been involved in the previous iteration – analysts, designers, programmers and testers – would participate.

6.2 Documentation

You should now be very familiar with the idea of documenting your Java code using comments. However documentation pervades the whole software development process. Imagine that a team is working on a software development project. The team members have different responsibilities: there are analysts, designers, user interface designers, programmers and others. Even with an effective project manager, good communication between the different people involved is a key success factor. Much of this communication is in the form of written documentation.

A programmer will not get far if they cannot understand what the designers have decided. Neither will the designers make progress if they cannot understand the work of the analysts. **Project documentation** describes the activities, decisions and outcomes of the different phases of the project.

Project documentation is used *during* a project for communication between developers. It is also a vital ingredient in enabling the operational software to be *maintained* successfully, and allowing aspects of to be reused in creating new software. Adapting

software simply by trying to understand and change the code alone is usually doomed to failure or, at best, leads to the production of code that is subsequently unintelligible.

The conclusions reached by each phase of development (including models such as sequence diagrams) obviously should be part of the project documentation. Other kinds of project information may also be relevant: a record of areas of debate and how differences of opinion were resolved, for example. In fact, any information that could potentially be of use to those maintaining the software, or to other developers working on similar projects, is relevant project documentation.

SAQ 10

Why might it be useful for the project documentation to include designs that were considered but discarded?

ANSWER.....

Discarded designs (as well as records of why they were discarded) can be useful to someone charged with modifying the software once it is in operation, or to someone working on a similar project, so that the reasons for design decisions can be understood and so that known pitfalls and blind alleys can be avoided.

6.3 Software tools

Software development teams often rely heavily on software tools, sometimes called **CASE (computer-aided software engineering) tools**. Javadoc, which you have used throughout the course, is an example of a CASE tool to aid documentation. Here are some examples of other kinds of tools, demonstrating the variety available.

Design tools

Design tools provide support for certain aspects of design. A design tool may incorporate a special drawing package which enables the formulation of designs using diagrams. There are many UML-based design tools.

Coding tools

Coding tools provide support for writing and running code. An example of a coding tool is an IDE (integrated development environment) such as BlueJ.

SAQ 11

What facilities might an IDE offer?

ANSWER.....

An IDE may offer the following:

- a specialised editor for writing and editing source code;
- facilities for checking the syntax and semantics of the source code;
- facilities for structuring programs into separate projects, and for creating repositories of associated documents;
- an integrated compiler.

Some tools offer integration and automation of elements of design, coding and testing. A tool might enable the user to specify aspects of the design, via UML diagrams for example, and then automatically produce corresponding outline program code. For example, you might produce a sequence diagram which the tool would take as the basis for generating skeletal outlines of methods. The more detailed the design, the more code is automatically generated.

CASE tools would appear to significantly reduce the work involved in the production of software. Consequently you might be surprised to learn that some developers prefer not to use them. There are several reasons for this:

1 Developers are forced to describe their designs in a format tightly prescribed by the tool – this may be inappropriate for some projects.

2 The overheads of getting to grips with a necessarily complex tool and working with its idiosyncrasies can be high.

3 Automatically produced code can be less readable and more complex than necessary. Furthermore such code may not adhere to a company's in-house conventions.

Exercise 5

Earlier in this unit we described a UML-type diagram as one that varies in some minor way from the specification set out in the UML standard.

Suppose a particular CASE tool produces outline code when it is given a design expressed in strict UML. Why would such a tool not generally accept a UML-type diagram instead?

Solution

A CASE tool is programmed to carry out certain processes (to produce the code) given specific input (a UML diagram). It will not be programmed to deal with other inputs such as even minor variations on strict UML.

Testing tools

There are many different kinds of **testing tool**. A **code-based testing tool** automatically analyses code and produces test cases ensuring that certain aspects of the code (for example, each path through it) are tested. A **test driver tool** executes the software being tested with specified inputs.

JUnit, which you used in *Unit 13*, is a tool incorporated into BlueJ that assists in the testing of Java classes. It enables the establishment of a testing framework specific to a program, then automatically performs tasks such as initialising objects for testing, and executing specified sets of tests.

7 Summary

This unit began by introducing you to the idea of *developing* software.

Through exploring the objects and collaborations at work in the School Management application, and using class diagrams, object diagrams and sequence diagrams for illustration, you learnt about the complexity that can be involved even in a relatively simple application.

Such complexity is managed by developing software in a systematic, progressive way, with interlinked phases of development and by using models. You were introduced to the phases and to the modelling language, UML, which enables you to produce consistent diagrammatic models that are an aid to communication between project members and to documenting the project.

Software development methods – ways of putting the phases of development together – are important when developing software. You were introduced to two of them: the waterfall and iterative methods.

The term software engineering is often used to describe the process of developing large-scale software projects in a way that is similar to engineering any large physical artefact. You were given the flavour of some of the elements of software engineering: e.g. how teams of developers work on a project, including the different team roles and the variety of tools used to assist in development.

This unit has provided only a brief introduction to the important ideas in software development. If you are interested in learning more about this subject, we suggest you investigate some of the other Open University computing courses that discuss the concepts, and techniques, of software engineering in more detail.

LEARNING OUTCOMES

After studying this unit you should be able to:

- describe the meaning of each of this unit's key terms (summarised in the Glossary);
- represent classes, and their inheritance relationships, and the links between their instances, using class diagrams;
- represent objects, and the links between them, using object diagrams;
- identify, by inspecting code, objects corresponding to real-world entities;
- identify, by inspecting code, how a link between objects is implemented;
- identify code corresponding to a design illustrated by a sequence diagram;
- explain why it is important to develop software systematically;
- outline what is involved in each of the following development phases:
  - requirements specification,
  - developing structural models,
  - designing dynamic models,
  - detailed design and implementation,
  - testing,
  - maintenance;
- describe the roles of diagrams, models and modelling languages in developing software;
- describe the reasons why UML has grown in importance as a modelling language for software development;
- outline what a software development method is and describe the essential features of the waterfall and iterative methods;
- describe some aspects of software engineering, i.e. different team roles and tools.

Glossary

abstraction A description that focuses on the essential features of a problem and ignores other details.

activation rectangle An element in a **sequence diagram** that represents a period during which a particular object is active.

adaptive method A method of **software development** which embraces change by building space for it into the schedule.

analysis In this context, analysis involves analysing the specified **requirements** to develop a detailed understanding, in computing terms, of what the software has to do. The outcome is a **requirements specification** document.

CASE (computer-aided software engineering) tool A software tool used to help in some aspect of **software development**.

client This term has two main meanings in the context of **software development**: (i) the object in a collaboration which requests a service; (ii) the person(s) commissioning software.

code-based testing tool A **testing tool** that automatically analyses code and produces test cases.

coding tool A **CASE tool** that aids writing or running code.

collaboration One object requesting a service from another object.

collaborator A participant in a collaboration.

design Design involves deciding how the software will meet the specified **requirements**.

design tool A **CASE tool** that aids some aspect of **design**.

designer A developer whose role is to work on the **design** stages of a project.

designing dynamic models Determining what interactions among objects will achieve the tasks required of the software.

detailed design and implementation Deciding what existing classes can be reused and what programming constructs are appropriate as well as writing the actual code.

developing a structural model Analysing the **requirements** to determine the classes and connections between them that are appropriate for the area the software is being written for, thus defining a structure for the software.

developing a user interface Designing the user interface and determining how it will communicate with the domain model.

domain model That part of the software that models the **problem domain** and is not directly concerned with how communication with the user is achieved.

dynamic model An illustration of events occurring in executing software over time.

identifier A label chosen to identify an object in the software.

implementation Translating the **design** into program code in some suitable programming language.

iteration One cycle through the phases involved in an **iterative method**.

iterative method An adaptive method of **software development** in which phases are repeated iteratively in a systematic manner.

lifeline An element in a **sequence diagram** that represents the time during which an object exists.

link A connection between two objects.

maintenance The phase of a software development process associated with keeping the software working to the satisfaction of its users.

modelling language A specification of how models should be constructed so that their meaning is unambiguous.

object diagram A diagram of objects and the **links** between them.

OMG (Object Management Group) A consortium of computing companies which sets standards across the software industry, including the **UML standard**.

phase A stage of **software development**.

predictive method A **software development** method that is largely based on the **waterfall method** and therefore benefits from simplicity of planning, and predictability.

problem domain The collection of real-world entities within the application area that exhibit the behaviours that the required software has to model.

programmer A developer whose role is to implement the code.

project documentation A written description of the activities, decisions and outcomes of a project's **phases**.

project failure A situation where a project fails to deliver the client's requirements.

project manager A person who plans and oversees the running of a software development project.

prototype An early working version of the software or part of it.

requirements What is required of the software.

requirements specification Eliciting and analysing what the client wants in order to produce a detailed and complete specification of the **requirements** of the software in terms of what it should do.

review A point within an iterative **software development method** where developers and clients can take changes into account.

sequence diagram An illustration of objects collaborating to carry out a particular task.

software developer An umbrella title, referring to someone who takes on one or more of a range of jobs within **software development**.

software development A planned, phased process, involving modelling different aspects of the software as well as implementing, **testing** and maintaining it.

software development method A particular set of development **phases** and activities, applied in a particular order.

software engineering A term used to refer to a wide range of concerns connected with carrying out systematic **software development**.

software model An illustration or description of the software, or of part of it, which emphasises certain aspects and omits others.

software tester A developer whose role is to test the software as it is being developed.

static model An illustration of the state of the software, or part of it, at a particular imagined time during execution.

strict UML diagram A diagram that adheres strictly to the **UML standard**.

systems analyst A developer who ideally has knowledge of both the **problem domain** (the client's business needs) and software design and whose role is to analyse the feasibility of proposed software and how it will impact on the client's business practices.

technical writer A developer whose role is to develop user documentation.

test driver tool A **testing tool** that executes the software being tested with specified inputs.

testing The activities that take place at each **phase** of development to ensure that what is produced relates correctly to the previous phase and to the **requirements**.

testing tool A **CASE tool** that aids some aspect of **testing**.

UML (Unified Modeling Language) A **modelling language** based on diagrams.

UML standard The currently accepted specification of what is valid **UML** and how it should be used.

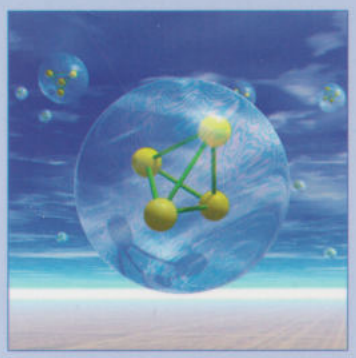
UML-type diagram A diagram that varies in some small way from the strict **UML standard**.

waterfall method A traditional and idealised view of developing software by strictly following a sequence of **phases**.

Acknowledgement

Figure 2: Map of the London Underground. Reproduced by permission of the London Transport Museum.

Index



M255 Unit 14
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

UNIT 14

Block 4

M255 Unit 14
ISBN 0 7492 1358 2